CS 4400
Computer Systems

LECTURE 23

Concurrent programming

Threads

Shared variables

Application-Level Concurrency

- Computing in parallel on multiprocessors
  - logical vs. physical concurrency
- Accessing slow I/O devices
  - already done by kernel, can be done at app-level
- Interacting with humans
  - create a separate concurrent flow to handle each user action
- Reducing latency by deferring work
  - defer coalescing to a concurrent flow that runs at low priority
- Service multiple network clients
  - create a separate concurrent flow for each client (more later)

Building Concurrent Programs

Three basic approaches:

- Processes
  - Each logical control flow is a processes (kernel-scheduled).
  - Separate virtual address spaces—requires explicit IPC.
- I/O multiplexing (see text)
  - Single process, all flows share same address space.
- Threads
  - Logical flows that run in the context of a single process.
  - Kernel schedules each thread.
  - Hybrid approach—kernel-scheduled, shared address space.

Running Example

- Echo server from 11.4.
- The client reads strings from a file until EOF.
  - Sends string to server, then prints string received from server.
- The server receives strings from and returns strings to the client until the connection with the client is closed.
- Functions from the Sockets Interface and the textbook's RIO package are used.
- We will look at making a concurrent version of this application using processes, and then threads.

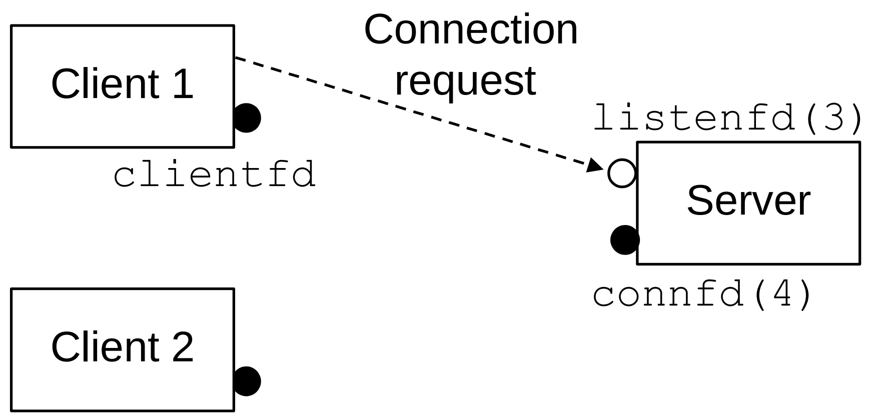
```
/* echoclient.c - an echo client */
#include "csapp.h"

int main(int argc, char** argv) {
  int clientfd;                /* client's socket descriptor */
  int port;                    /* server's well-known port */
  char* host;                  /* server's host name */
  char buf[MAXLINE];           /* input string */
  rio_t rio;                   /* read buffer */

  if(argc != 3)
    /* ERROR, QUIT */
  host = argv[1];
  port = atoi(argv[2]);

  clientfd = Open_clientfd(host, port);  /* connect to server */
  Rio_readinitb(&rio, clientfd);         /* init read buffer */

  while (Fgets(buf, MAXLINE, stdin) != NULL) {
    /* send input string to server */
    Rio_writen(clientfd, buf, strlen(buf));
    /* receive string from server */
    Rio_readlineb(&rio, buf, MAXLINE);
    Fputs(buf, stdout);   /* print string */
  }
  Close(clientfd);          /* close connection to server */
  exit(0);
}
```

```
/* echoserveri.c - an iterative echo server */
#include "csapp.h"

void echo(int connfd);

int main(int argc, char** argv) {
  int listenfd;             /* “listening” socket descriptor */
  int connfd;               /* “connected” socket descriptor */
  int port;                 /* server's well-known port */
  int clientlen;            /* length of client's address */
  struct sockaddr_in clientaddr;  /* client's address */

  if(argc != 2)
    /* ERROR, QUIT */
  port = atoi(argv[1]);

  listenfd = Open_listenfd(port);   /* listen for client */
  while (1) {
    clientlen = sizeof(clientaddr);
    /* connect to client */
    connfd = Accept(listenfd, (SA*)&clientaddr, &clientlen);
    /* service the client */
    echo(connfd);
    /* close connection to client */
    Close(connfd);
  }
    exit(0);
}
```

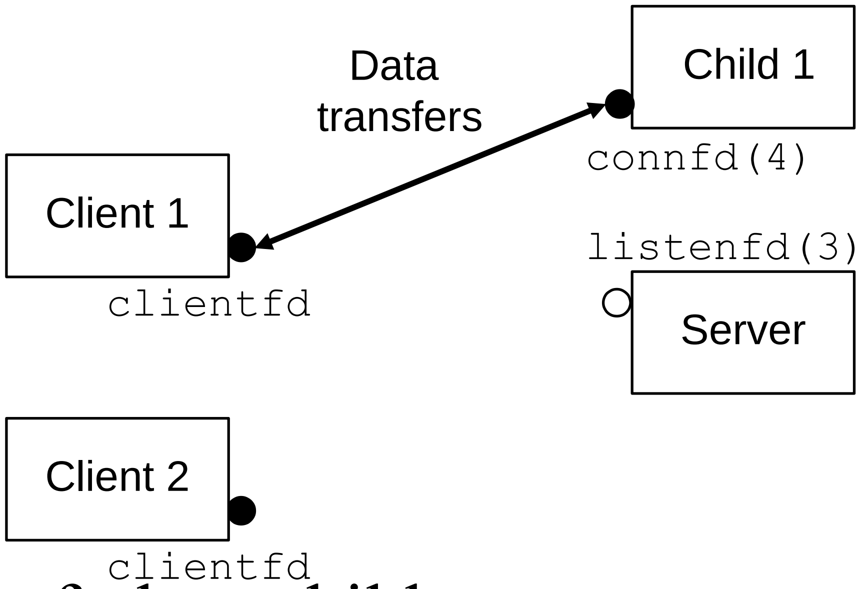
```
/* echo.c - read and echo text lines until client closes
            connection */
#include "csapp.h"

void echo(int connfd) {
  char buf[MAXLINE];          /* input string */
  rio_t rio;                  /* read buffer */

  Rio_readinitb(&rio, connfd);   /* init read buffer */

  /* while connection with client remains open . . . */
  while(Rio_readlineb(&rio, buf, MAXLINE) != 0) {
    /* read string from client and send string back to client */
    Rio_writen(connfd, buf, n);
  }
}
```

Concurrency w/ Processes

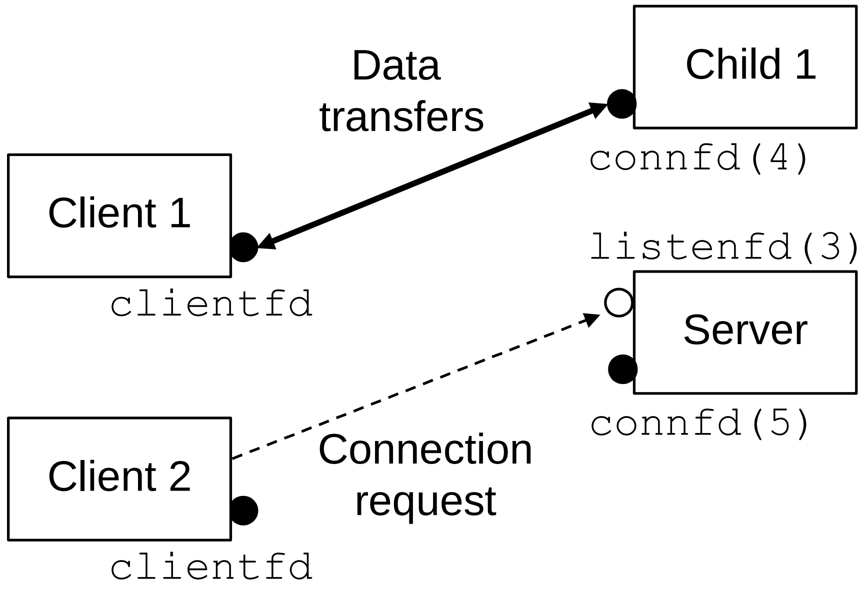
- Accept client connection requests in parent, and then create a new child process to service each new client.
- *Example*: two clients, one server listening on descriptor 3



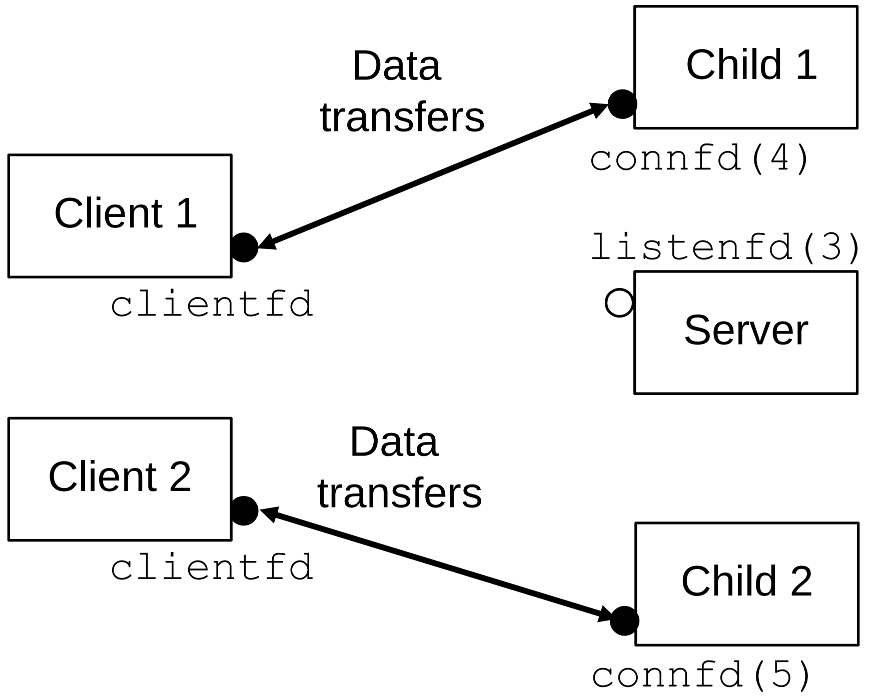
Step 1: Server accepts connection request from client.



Step 2: Server forks a child process to service the client.



Step 3: Server accepts another connection request.



Step 4: Server forks another child to service new client.

(Parent is waiting for next connection request and two children are servicing their respective clients concurrently.)

It is critical for the parent to close its copy of `connfd` after a `fork()`. Why?

```c
/* echoserverp.c - a concurrent echo server (based on processes) */
#include "csapp.h"
void echo(int connfd);

void sigchld_handler(int sig) {
  while(waitpid(-1, 0, WNOHANG) > 0) ;   /* reap zombie children */
  return;
}

int main(int argc, char** argv) {
  int listenfd, connfd, port, clientlen=sizeof(struct sockaddr_in);
  struct sockaddr_in clientaddr;

  if(argc != 2)
    /* ERROR, QUIT */
  port = atoi(argv[1]);

  Signal(SIGCHLD, sigchld_handler);
  listenfd = Open_listenfd(port);
  while (1) {
    connfd = Accept(listenfd, (SA*)&clientaddr, &clientlen);
    if(Fork() == 0) {
      Close(listenfd); /* Child closes its listening socket */
      echo(connfd);    /* Child services client */
      Close(connfd);   /* Child closes connection with client */
      exit(0);         /* Child exits */
    }
    Close(connfd); /* Parent closes connected socket (important!) */
    /* Connection to client will not be terminated until both parent's
       and child's copies of connfd are closed (potential mem leak). */
  }
}
```

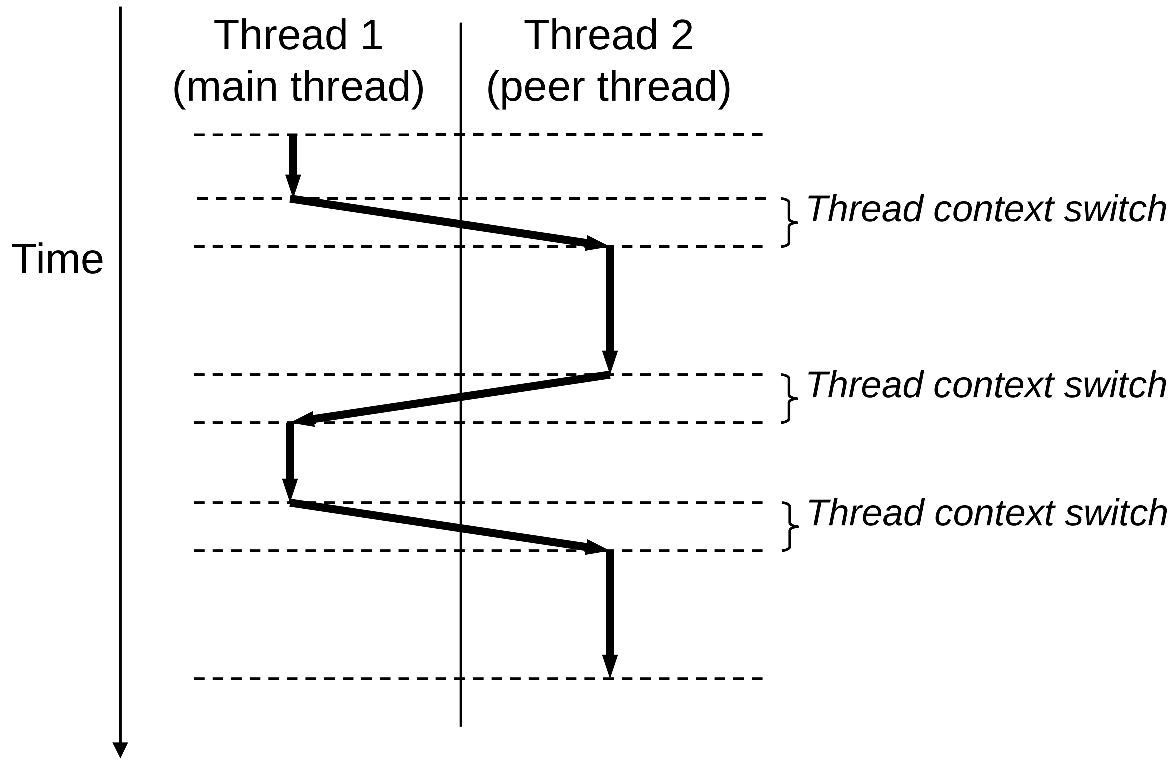
Pros/Cons of Processes

- *Pro*: Clean model for sharing state information between parents and children.
  - file tables are shared (child gets copy of socket descriptors)
  - user address spaces are not shared (cannot overwrite virtual memory of another process)
- *Con*: Separate address spaces make it more difficult for processes to share state information.
  - must use explicit interprocess communications (IPC) mechanisms
  - examples of explicit IPC?

Threads

- A *thread* is a logical flow that runs in the context of a process.
  - so far, our programs have consisted of a single thread
- The kernel automatically schedules threads.
- Each thread has its own *thread context*.
  - thread ID (TID)—a unique integer
  - stack and stack pointer
  - program counter, gen-purpose registers, and condition codes
- All threads running in a process share the entire virtual address space.

Execution Model



Control passes to the peer thread because the main thread executes a slow system call or is interrupted by the system's interval timer.

- Each process begins life as a single, *main thread*.
- The main thread creates a *peer thread*, and from that point the two threads run concurrently.

Threads vs. Processes

- A thread context switch is faster than a process context switch.
  - a thread context is much smaller than a process context
- Threads are not organized in a rigid parent-child hierarchy.
  - threads associated with a process form a pool of peers, independent of which threads were created by which other threads
  - a thread can kill any of its peers, or wait for any of its peers to terminate
  - each peer can read or write the same shared data

Example: Pthreads

```
/* Pthreads is a standard interface for manipulating threads from C
   programs. */

#include "csapp.h"
void* thread(void* vargp);

/* main thread */
int main() {
  pthread_t tid;   /* thread ID of peer thread */
  /* create peer thread */
  Pthread_create(&tid, NULL, thread, NULL);
  /*--Now, main thread and peer thread are running concurrently.--*/
  /* wait for peer thread to terminate */
  Pthread_join(tid, NULL);
  /* terminate all threads */
  exit(0);
}

/* The code and local data for a thread are encapsulated in a thread
   routine.  Each thread routine takes as input a single generic
    pointer and returns a generic pointer. */
void* thread(void* vargp) {
  printf("Hello, world!\n");
  return NULL;   /* terminate peer thread */
}
```

Creating Threads

```
typedef void* (func)(void*);
int pthread_create(pthread_t* tid,
        pthread_attr_t* attr, func* f, void* arg);
```

- Creates a new thread and runs the thread routine `f` in the context of the new thread and with input argument `arg`.
- `attr` can be used to change the default thread attributes.
  - we'll always use `NULL`
- Upon return, `tid` is set to the ID of the new thread.
- A thread can determine its own thread ID using:

```
pthread_t pthread_self(void);
```

Terminating Threads

A thread terminates in one of the following ways.

- Its top-level thread routine returns.
- `int pthread_exit(void* thread_return);`
  - if called by the main thread, it waits for all peer threads
- Calls `exit`, which terminates the process and all associated threads.
- Another peer thread calls `pthread_cancel` with the ID of the current thread.

```
int pthread_cancel(pthread_t tid);
```

Reaping Terminated Threads

```
int pthread_join(pthread_t tid,
                 void** thread_return);
```

- Blocks until thread `tid` terminates.
- Assigns the `void*` returned by the thread routine to the location pointed to by `thread_return`.
- Reaps any memory resources held by the terminated thread.
- Unlike `wait_pid`, this function can only wait for a specific thread to terminate.

Detaching Threads

- At any time, a thread is joinable or detached.
  - *joinable*—the thread can be reaped and killed by other threads, at which time its memory resources are freed
  - *detached*—the thread cannot be reaped or killed by other threads, and its memory resources are free automatically by the system when it terminates
- By default, all threads are created joinable.
- To avoid memory leaks, each joinable thread should either be reaped by another thread or detached.

```
int pthread_detach(pthread_t tid);
```

```c
/* echoservert.c - a concurrent echo server using threads */
#include "csapp.h"

void echo(int connfd);
void* thread(void* vargp);

int main(int argc, char** argv) {
  int listenfd, *connfdp, port, clientlen=sizeof(struct sockaddr_in);
  struct sockaddr_in clientaddr;
  pthread_t tid;

  if(argc != 2)
    /* ERROR, QUIT */
  port = atoi(argv[1]);

  listenfd = Open_listenfd(port);
  while(1) {
    connfdp = Malloc(sizeof(int));   /* avoids a race (more next) */
    *connfdp = Accept(listenfd, (SA*)&clientaddr, &clientlen);
    Pthread_create(&tid, NULL, thread, connfdp);
  }
}

void* thread(void* vargp) {
  int connfd = *((int*)vargp);
  Pthread_detach(pthread_self());  /* because threads are not being */
  Free(vargp);                     /* explicitly reaped */
  echo(connfd);
  Close(connfd);
  return NULL;
}
```

Avoiding a Race

- Incorrect approach:

```
while (1) {
  int connfd = Accept(...);
  Pthread_create(&tid, NULL, thread, &connfd);
  ...
}
void* thread(void* vargp) {
  int connfd = *((int*)vargp); ...
}
```

- Introduces a *race* between the assignment in the peer thread and the *next* `accept` call in the main thread.
- If peer thread's assignment occurs first: works.
- If main thread's `accept` call occurs first: doesn't work.

Threads Memory Model

- Each thread has its own separate thread context.
  - TID, stack, SP, PC, condition codes, and gen-purpose regs
- Each thread shares the rest of the process context with other threads.
  - code, read/write data, the heap, any shared library code/data, and the set of open files
  - if a shared memory location is modified by one thread, the other threads see the change (if they read the memory loc)
- While thread stacks are usually accessed independently by their respective threads, this is not a guarantee. Why?

Mapping Variables to Memory

- *Global variable*—any variable declared outside of a function.
  - one instance that can be referenced by any thread
- *Local automatic variables*—any variable declared inside a function without the `static` attribute.
  - each thread's stack contains its own instance (even if multiple threads execute the same thread routine)
- *Local static variables*—any variable declared inside a function with the `static` attribute.
  - like global variables, one instance for all threads

Example: Shared Variables

```
/* A variable is “shared” iff one of its instances is referenced by
   more than one thread. */
#include "csapp.h"
void* thread(void* vargp);

char** ptr;        /* global variable for all threads */

int main() {
  int i;           /* the main thread's local auto vars */
  pthread_t tid;
  char* msgs[N] = {"Hello from foo", "Hello from bar"};

  ptr = msgs;
  for(i = 0; i < 2; i++)   /* create two peer threads */
    Pthread_create(&tid, NULL, thread, (void*)i);
  Pthread_exit(NULL);
}

void* thread(void* vargp) {
  int myid = (int)vargp;    /* a local auto var for each peer thread */
  static int cnt = 0;   /* ONE local static var for all peer threads */
  printf("[%d]: %s (cnt=%d)\n", myid, ptr[myid], ++cnt);
}
```

Exercise: Shared Variables

Let `v.t` denote that an instance of variable `v` that resides on the local stack for thread `t`, which is either `m` (main), `p0` (peer 0), or `p1` (peer 1).

- For each variable instance `ptr`, `cnt`, `i.m`, `msgs.m`, `myid.p0`, and `myid.p1`:
  ref by `m`? ref by `p0`? ref by `p1`?
- Which of the variables `ptr`, `cnt`, `i`, `msgs`, and `myid` are shared?

```
/* badcnt.c */
#include "csapp.h"

#define NITERS 200000000
void* count(void* arg);

unsigned int cnt = 0;     /* shared counter variable */

int main() {
  pthread_t tid1, tid2;

  Pthread_create(&tid1, NULL, count, NULL);
  Pthread_create(&tid2, NULL, count, NULL);
  Pthread_join(tid1, NULL);
  Pthread_join(tid2, NULL);

  if(cnt != (unsigned)NITERS*2)
    printf("BOOM! cnt=%d\n", cnt);
  else
    printf("OK cnt=%d\n", cnt);
  exit(0);
}

/* thread routine */
void* count(void* arg) {
  int i;
  for(i = 0; i < NITERS; i++)
    cnt++;
  return NULL;
}
```

```
unix> ./badcnt
BOOM! ctr=278125352

unix> ./badcnt
BOOM! ctr=271726247

unix> ./badcnt
BOOM! ctr=276537330
```

Example: Synchronization Error

C code for thread i

```
for (i=0; i<NITERS; i++)
    ctr++;
```

Asm code for thread i

```
.L9:
    movl -4(%ebp),%eax
    cmpl $99999999,%eax
    jle .L12
    jmp .L10
```

H_i : Head

```
.L12:
    movl ctr,%eax
    leal 1(%eax),%edx
    movl %edx,ctr
```

L_i : Load `ctr`

U_i : Update `ctr`

S_i : Store `ctr`

```
.L11:
    movl -4(%ebp),%eax
    leal 1(%eax),%edx
    movl %edx,-4(%ebp)
    jmp .L9
.L10:
```

T_i : Tail

- H_i and T_i manipulate only local stack variables.
- L_i U_i and S_i manipulate the shared counter variable.

Example: A Successful Ordering

Step	Thread	Instr	`%eax`$_1$	`%eax`$_2$	`cnt`
1	1	H_1	--	--	0
2	1	L_1	0	--	0
3	1	U_1	1	--	0
4	1	S_1	1	--	1
5	2	H_2	--	--	1
6	2	L_2	--	1	1
7	2	U_2	--	2	1
8	2	S_2	--	2	2
9	2	T_2	--	2	2
10	1	T_1	1	--	2

The OS will choose an interleaving of the instructions in the two threads.

Example: An Unsuccesful Ordering

Step	Thread	Instr	`%eax`$_1$	`%eax`$_2$	`cnt`
1	1	H_1	--	--	0
2	1	L_1	0	--	0
3	1	U_1	1	--	0
4	2	H_2	--	--	0
5	2	L_2	--	0	0
6	1	S_1	1	--	1
7	1	T_1	1	--	1
8	2	U_2	--	1	1
9	2	S_2	--	1	1
10	2	T_2	--	1	1

There is no way to predict whether the OS will choose a correct ordering of threads.