Outline

- Chapter 3: Processes
  - So far -
    - Processes are programs in execution
      - Kernel keeps track of them using process control blocks
      - PCBs are saved and restored at context switch
    - Schedulers choose the ready process to run (more in Ch 5)
  - Next -
    - Processes create other processes
      - On exit, status returned to parent
    - Processes communicate with each other using shared memory or message passing
- Chapter 4: Threads

Operations on processes

- Process creation
  - Parent creates new process forming a tree
  - Child process can run concurrently with parent or not
  - Child can share all resources, some or none at all
- Process termination
  - Exit for normal termination
    - Output data from child to parent (via **wait**)
    - exit() and _exit() functions
  - Abort for abnormal kernel initiated termination
  - Some OS require the presence of parent to allow child

C example of fork

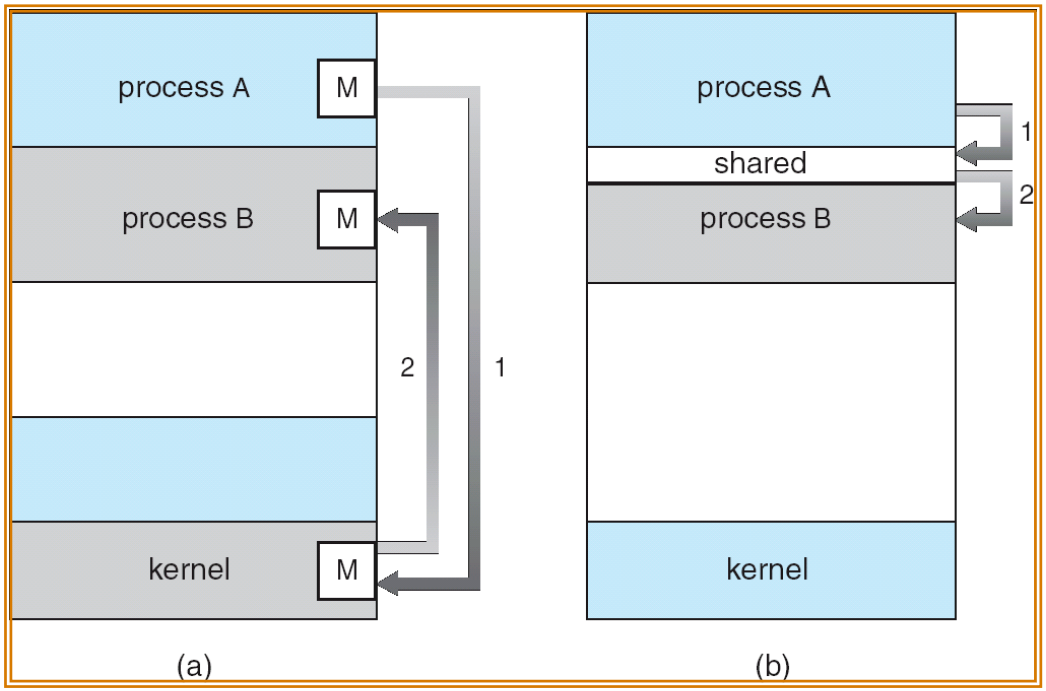
```
int main()
   {
        pid_t  pid;
        /* fork another process */
        pid = fork();
        if (pid < 0) { /* error occurred */
               fprintf(stderr, "Fork Failed");
               exit(-1);
        }
        else if (pid == 0) { /* child process */
               execlp("/bin/ls", "ls", NULL);
        }
        else { /* parent process */
               /* waits for child to complete */
               wait (NULL);
               printf ("Child Complete");
               exit(0);
        }
   }
```

Interprocess communications

- **Independent** process cannot affect or be affected by the execution of another process
- **Cooperating** process can affect or be affected by the execution of another process
- Advantages of process cooperation
  - Information sharing
  - Computation speed-up
  - Modularity
  - Convenience

IPC mechanisms

- Shared memory
  - Create shared memory region
  - When one process writes into this region, the other process can see it and vice versa
- Message passing
  - Explicitly send() and receive()



Producer/consumer using shared memory

- Shared data

```
#define BUFFER_SIZE 10
typedef struct {
   . . .
} item;

item buffer[BUFFER_SIZE];
int in = 0;
int out = 0;
```

- Solution is correct, but can only use BUFFER_SIZE-1 elements

Insert/Remove methods

```
while (true) {
/* Produce an item */
 while (((in = (in + 1) % BUFFER SIZE count)  == out)
        ;   /* do nothing -- no free buffers */
   buffer[in] = item;
   in = (in + 1) % BUFFER SIZE;
}


while (true) {
   while (in == out)
      ; // do nothing -- nothing to consume

   // remove an item from the buffer
   item = buffer[out];
   out = (out + 1) % BUFFER SIZE;
   return item;
}
```

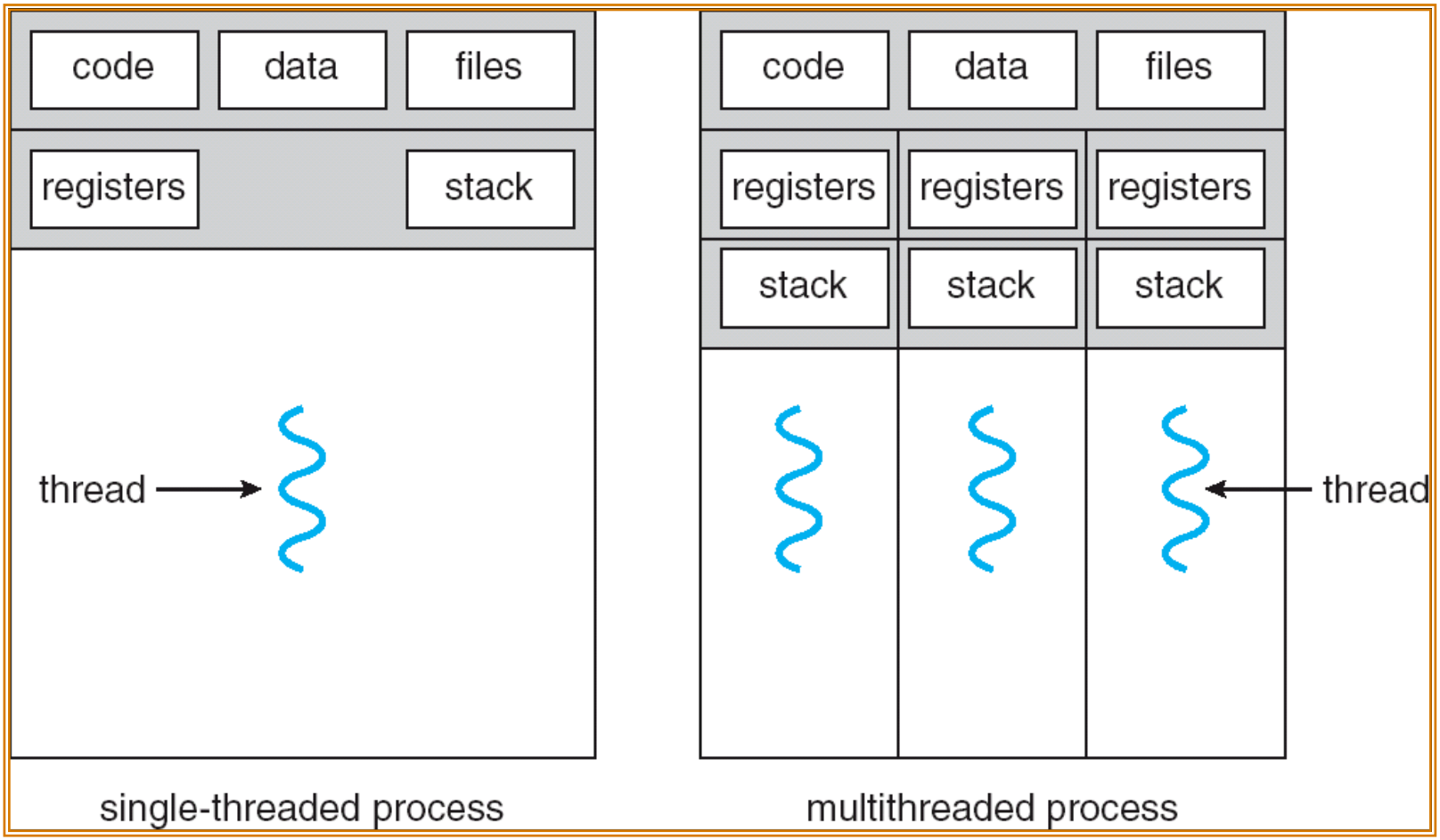
Message passing

- Requires ways to name objects (same machine or different machine).
- Communications can be synchronous or asynchronous.
- May need to buffer messages that are not ready to be read

Chapter 4: Threads

- Thread is the basic unit of CPU utilization. So far, our implicit assumption was that each process has a single thread of execution. However, each process can have multiple threads of execution, potentially working on more than one thing at the same time
- Threads in the same process share text, data, open files, signals and other resources. Each thread has its own execution context and stack.

Single and Multithreaded Processes



Sample pthreads program

```
void *add_runner(void *param){
  int upper = atoi(param);

   for (int i = 1; i <= upper; i++)
       sum += i;

  pthread_exit(0); }
```

```
void *sub_runner(void *param){
  int upper = atoi(param);

  for (int i = 1; i <= upper; i++)
       sum -= i;

  pthread_exit(0); }
```

Sample pthreads library

```
int sum; /* this data is shared by the thread(s) */
main(int argc, char *argv[]) {
   pthread_t tid; /* the thread identifier */
   pthread_attr_t attr; /* set of attributes for the thread */

   pthread_attr_init(&attr); /* get the default attributes */
   pthread_create(&tid, &attr, add_runner, argv[1]);
   pthread_create(&tid, &attr, sub_runner, argv[1]);

   for (int i = 1; i <= 50; i++)
     printf("sum = %d\n",sum);
   pthread_join(tid,NULL);
   printf("final sum = %d\n",sum); }
```

Benefits

- Responsiveness - Interactive applications can be performing two tasks at the same time (rendering, spell checking)
- Resource Sharing - Sharing resources between threads is easy (too easy?)
- Economy - Resource allocation between threads is fast (no protection issues)
- Utilization of MP Architectures - seamlessly assign multiple threads to multiple processors (if available). Future appears to be multi-core anyway.

Thread types

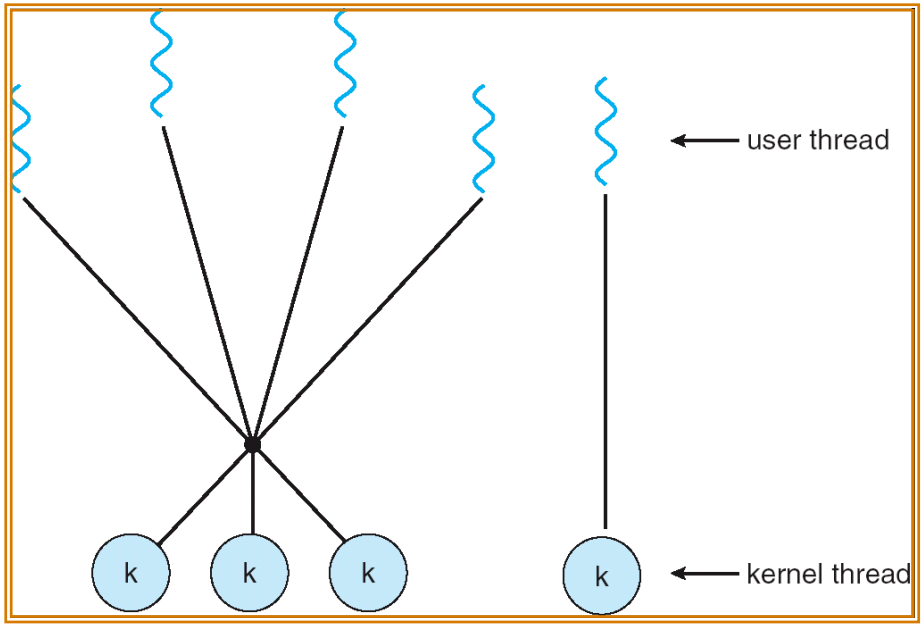
- User threads: thread management done by user-level threads library. Kernel does not know about these threads
  - Three primary thread libraries:
    - POSIX Pthreads
    - Win32 threads
    - Java threads
- Kernel threads: Supported by the Kernel and so more overhead than user threads
  - Examples: Windows XP/2000, Solaris, Linux, Mac OS X
- User threads map into kernel threads

Multithreading Models

- Many-to-One: Many user-level threads mapped to single kernel thread
  - If a thread blocks inside kernel, all the other threads cannot run
  - Examples: Solaris Green Threads, GNU Pthreads
- One-to-One: Each user-level thread maps to kernel thread
- Many-to-Many: Allows many user level threads to be mapped to many kernel threads
  - Allows the operating system to create a sufficient number of kernel threads

Two-level Model

- Similar to M:M, except that it allows a user thread to be bound to kernel thread
- Examples
  - IRIX
  - HP-UX
  - Tru64 UNIX
  - Solaris 8 and earlier



Threading issues

- What happens if a thread invokes fork() or exec()?
  - Unixes support two fork() functions, once which creates a new process with all threads and one with a single threaded new process
- Thread cancellation
  - More tricky than process killing. Thread might be in the middle of something.
    - Asynchronous
    - Deferred cancellation: Target thread periodically checks to see if it had been cancelled
- Signal handling: who should get a signal?
  - E.g., pressing <Ctrl>-C (interrupt) or Divide-by-zero
  - Thread to which signal applies, every thread, certain threads or a specific thread to receive signals?

Issues (cont)

- Thread pools:
  - Web servers pre-allocate threads and have these threads wait for new requests. Amortize the cost of thread creation and bound the number of threads