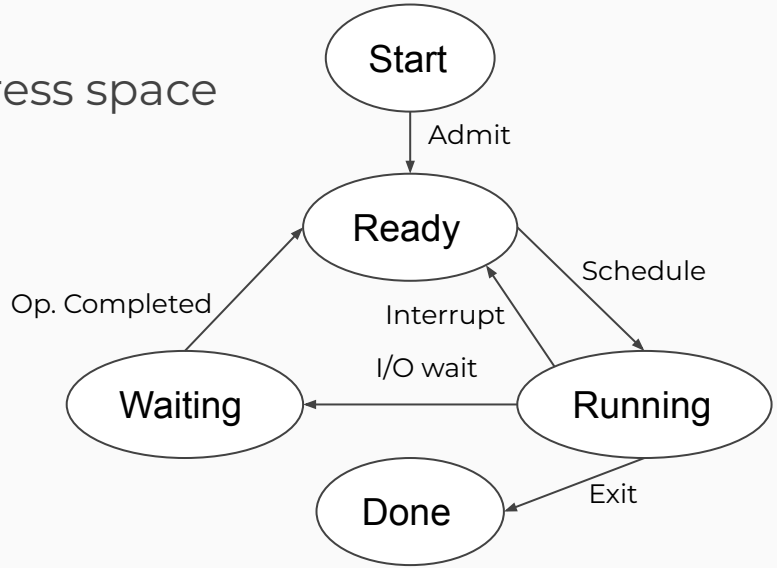
CS 330 - Operating Systems

Processes and Threads

12-08-2025

Process

- Execution environment with restricted rights
  - Thread(s) + address space
  - Encapsulate one or more threads sharing process resources
- Has similar state as threads
  - PC, SP, registers along with address space
- States



Process Control Block

```
struct context {
  uint edi;
  uint esi;
  uint ebx;
  uint ebp;
  uint eip;
};

enum procstate { UNUSED, EMBRYO, SLEEPING, RUNNABLE, RUNNING, ZOMBIE };

struct proc {
  uint sz;                     // Size of process memory (bytes)
  pde_t* pgdir;                // Page table
  char *kstack;                // Bottom of kernel stack for this process
  enum procstate state;        // Process state
  volatile int pid;            // Process ID
  struct proc *parent;         // Parent process
  struct trapframe *tf;        // Trap frame for current syscall
  struct context *context;     // swtch() here to run process
  void *chan;                  // If non-zero, sleeping on chan
  int killed;                  // If non-zero, have been killed
  struct file *ofile[NOFILE];  // Open files
  struct inode *cwd;           // Current directory
  struct shared *shared;       // Shared memory record (0 -> none)
  char name[16];               // Process name (debugging)
};
```

Process APIs in UNIX Systems

- fork()
  - fork() system call is used to create a new process
- exec()
  - if you want to run a different program (not the one in current)
- wait()
  - parent to wait for a child process to finish what it has been doing
- kill()
  - kills the process specified by the identifier

fork()

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main(int argc, char *argv[]) {
  printf("hello world (pid:%d)\n", (int) getpid());
  int rc = fork();
  if (rc < 0) {
    // fork failed
    fprintf(stderr, "fork failed\n");
    exit(1);
  } else if (rc == 0) {
    // child (new process)
    printf("hello, I am child (pid:%d)\n", (int) getpid());
  } else {
    // parent goes down this path (main)
    printf("hello, I am parent of %d (pid:%d)\n",
           rc, (int) getpid());
  }
  return 0;
}
```

wait()

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>

int main(int argc, char *argv[]) {
  printf("hello world (pid:%d)\n", (int) getpid());
  int rc = fork();
  if (rc < 0) {          // fork failed; exit
    fprintf(stderr, "fork failed\n");
    exit(1);
  } else if (rc == 0) { // child (new process)
    printf("hello, I am child (pid:%d)\n", (int) getpid());
  } else {               // parent goes down this path (main)
    int rc_wait = wait(NULL);
    printf("hello, I am parent of %d (rc_wait:%d) (pid:%d)\n",
           rc, rc_wait, (int) getpid());
  }
  return 0;
}
```

exec()

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <sys/wait.h>

int main(int argc, char *argv[]) {
  printf("hello world (pid:%d)\n", (int) getpid());
  int rc = fork();
  if (rc < 0) {          // fork failed; exit
    fprintf(stderr, "fork failed\n");
    exit(1);
  } else if (rc == 0) { // child (new process)
    printf("hello, I am child (pid:%d)\n", (int) getpid());
    char *myargs[3];
    myargs[0] = strdup("wc");   // program: "wc" (word count)
    myargs[1] = strdup("p3.c"); // argument: file to count
    myargs[2] = NULL;           // marks end of array
    execvp(myargs[0], myargs);  // runs word count
    printf("this shouldn't print out");
  } else {               // parent goes down this path (main)
    int rc_wait = wait(NULL);
    printf("hello, I am parent of %d (rc_wait:%d) (pid:%d)\n",
           rc, rc_wait, (int) getpid());
  }
  return 0;
}
```

Thread APIs

int **pthread_create**(
    pthread_t *thread,
    const pthread_attr_t *attr,
    void *(*start_routine)(void*),
    void *arg);

int **pthread_join**(
    pthread_t thread,
    void **value_ptr);

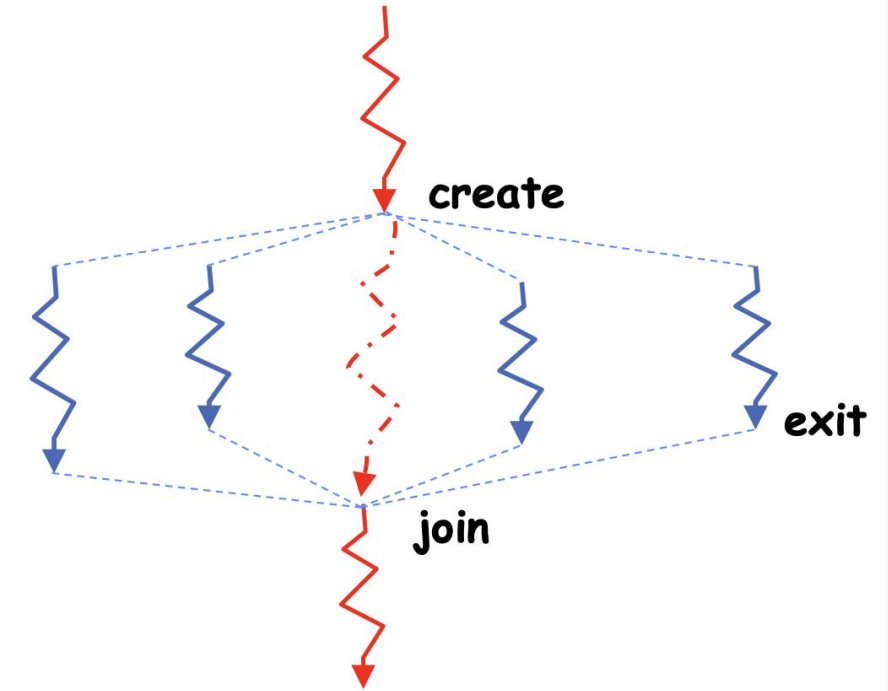
```c
#include <stdio.h>
#include <assert.h>
#include <pthread.h>
#include "common.h"
#include "common_threads.h"

void *mythread(void *arg) {
    printf("%s\n", (char *) arg);
    return NULL;
}

int
main(int argc, char *argv[]) {
    pthread_t p1, p2;
    int rc;
    printf("main: begin\n");
    Pthread_create(&p1, NULL, mythread, "A");
    Pthread_create(&p2, NULL, mythread, "B");
    // join waits for the threads to finish
    Pthread_join(p1, NULL);
    Pthread_join(p2, NULL);
    printf("main: end\n");
    return 0;
}
```

Thread APIs

- Main thread creates (forks) collection of sub-threads passing them args to work on...
- ... and then joins with them, collecting results.



Thread APIs

Possible Execution Traces
(Run t0)

- How many threads are there?
- How does the execution happen?

```
1  #include <stdio.h>
2  #include <assert.h>
3  #include <pthread.h>
4  #include "common.h"
5  #include "common_threads.h"
6
7  void *mythread(void *arg) {
8      printf("%s\n", (char *) arg);
9      return NULL;
10 }
11
12 int
13 main(int argc, char *argv[]) {
14     pthread_t p1, p2;
15     int rc;
16     printf("main: begin\n");
17     Pthread_create(&p1, NULL, mythread, "A");
18     Pthread_create(&p2, NULL, mythread, "B");
19     // join waits for the threads to finish
20     Pthread_join(p1, NULL);
21     Pthread_join(p2, NULL);
22     printf("main: end\n");
23     return 0;
24 }
```

Example with Threads

```
main() {
    create_thread(ComputePI, “pi.txt”);
    create_thread(PrintClassList, “classlist.txt”);
}
```

- create_thread
  - Spawns a new thread running the given procedure
  - Should behave as if another CPU is running the given procedure

Example with Threads

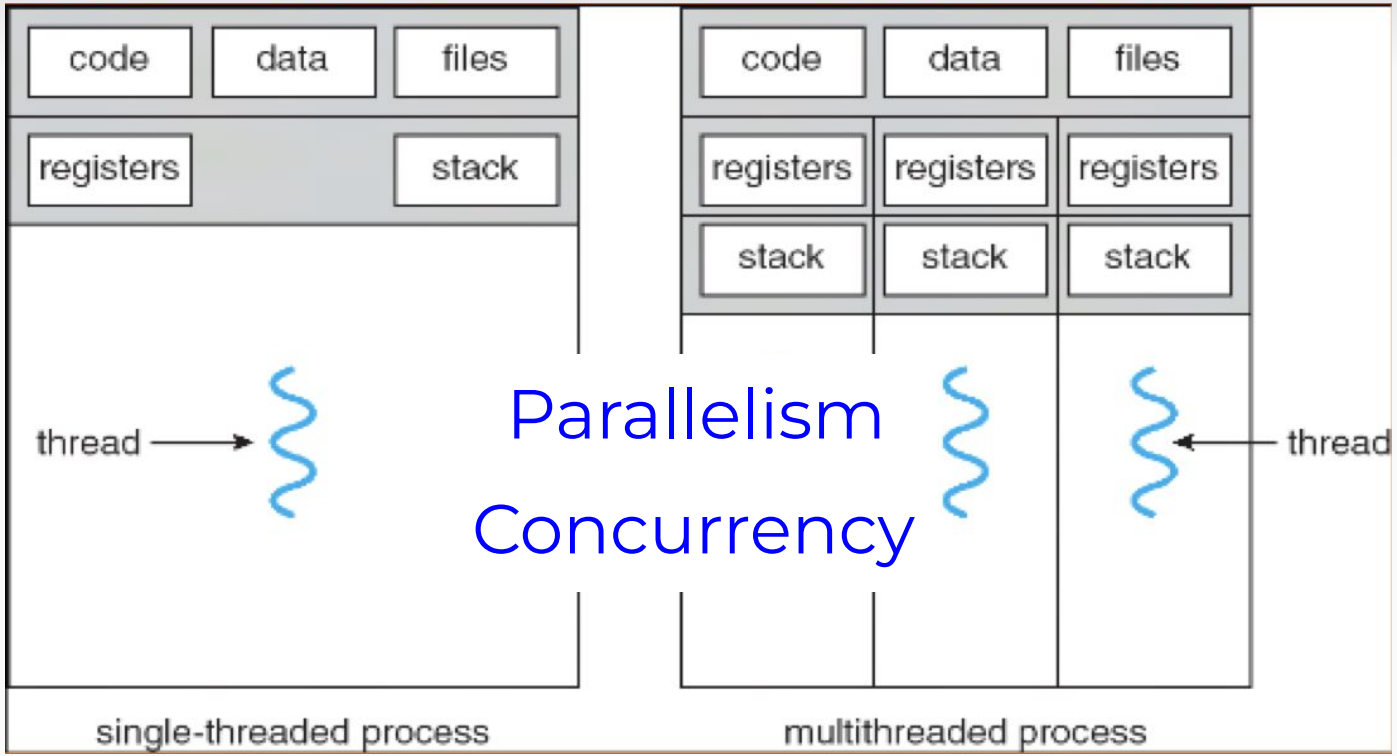
```
#include <stdio.h>
#include <stdlib.h>
#include "common.h"
#include "common_threads.h"

volatile int counter = 0;
int loops;

void *worker(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        counter++;
    }
    return NULL;
}
```

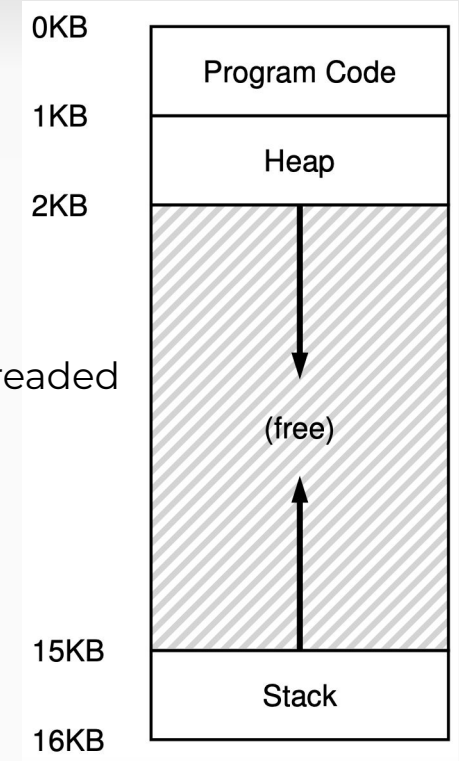
```
int main(int argc, char *argv[]) {
    if (argc != 2) {
        fprintf(stderr, "usage: threads <loops>\n");
        exit(1);
    }
    loops = atoi(argv[1]);
    pthread_t p1, p2;
    printf("Initial value : %d\n", counter);
    Pthread_create(&p1, NULL, worker, NULL);
    Pthread_create(&p2, NULL, worker, NULL);
    Pthread_join(p1, NULL);
    Pthread_join(p2, NULL);
    printf("Final value   : %d\n", counter);
    return 0;
}
```

Multithreading

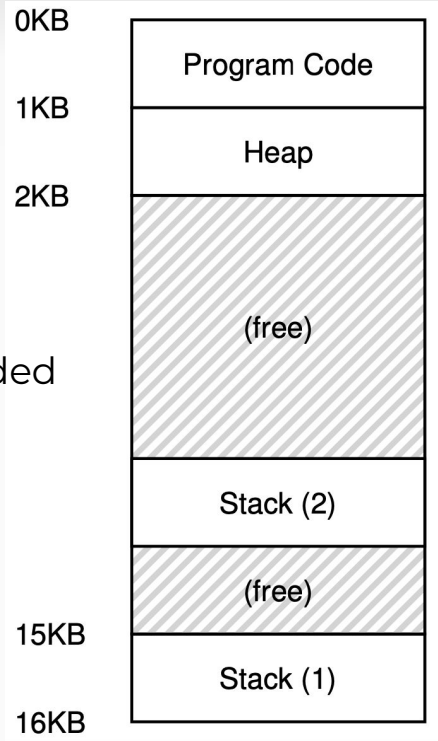


Multithreading

Single-threaded



Multi-threaded



Multiple processes

- Context-switch