lecture 13

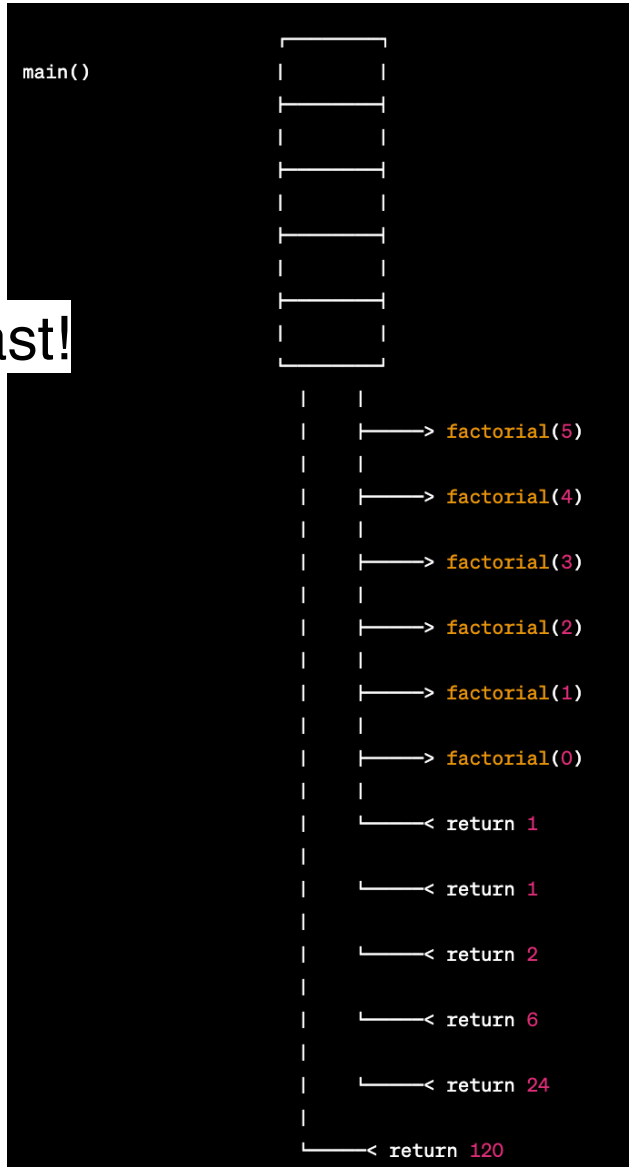
- *recursive functions*
- *ipc (using a common file in the file system to pass messages)*
- *mmap (improving the shared file approach - segue to shared memory)*
- *C++ threads into*

Kenneth J. Roche
University of Washington
AMATH 483 / 583 High Performance Scientific Computing
Spring Quarter 2025

- recursive functions

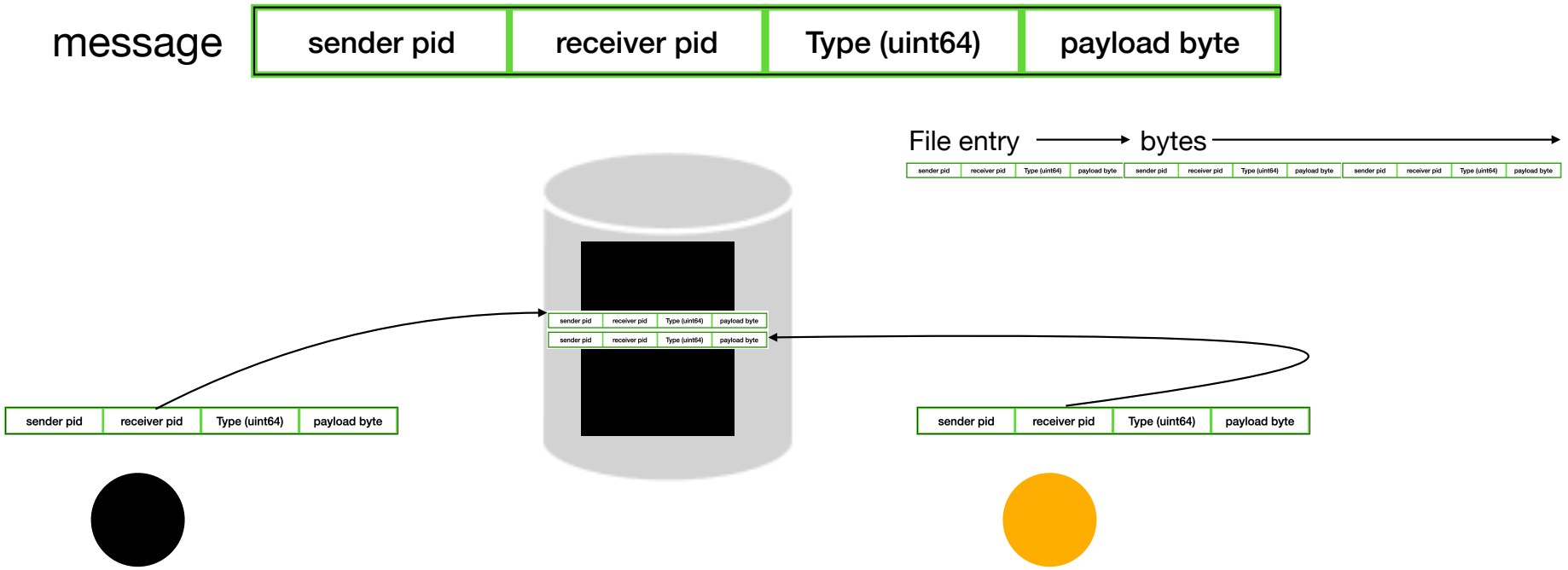
```cpp
#include <iostream>

int factorial(int n) {
    if (n == 0) {
        return 1;
    } else {
        return n * factorial(n - 1);
    }
}

int main() {
    std::cout << factorial(5) << std::endl;
    return 0;
}
```

stack grows fast!



enneth J. Roche
niversity of Washington
MATH 483 / 583 High Performance Scientific Computing
pring Quarter 2025

- *ipc (using a common file in the file system to pass messages)*




```
ssize_t write(int fd, const void *buf, size_t count);
```

Kenneth J. Roche
University of Washington
AMATH 483 / 583 High Performance Scientific Computing
Spring Quarter 2025

- *ipc (using a common file in the file system to pass messages)*

send message := write

```cpp
// send / recv example
#include <iostream>
#include <fstream>
#include <string>
#include <unistd.h>
#include <vector>
#include <fcntl.h>
#include <sys/file.h>
#include <sys/wait.h>

const std::string COMM_FILENAME = "comm_file.dat";

struct MessageHeader
{
    pid_t sender;
    pid_t receiver;
    uint64_t size;
};

void send_message(pid_t my_pid, pid_t target_pid, const std::string &message)
{
    // note I am using file descriptors here (integer file handles)
    int fd = open(COMM_FILENAME.c_str(), O_WRONLY | O_APPEND);
    if (fd == -1)
    {
        std::cerr << "Failed to open communication file for sending.\n";
        return;
    }

    flock(fd, LOCK_EX); // Exclusive lock

    MessageHeader header = {my_pid, target_pid, static_cast<uint64_t>(message.size())};
    write(fd, &header, sizeof(header));
    write(fd, message.data(), message.size());

    flock(fd, LOCK_UN); // Unlock
    close(fd);
}
```

- *ipc (using a common file in the file system to pass messages)*

receive message := read

```
42  std::vector<std::pair<pid_t, std::string>> receive_messages(pid_t my_pid)
43  {
44      std::vector<std::pair<pid_t, std::string>> inbox;
45
46      std::ifstream ifs(COMM_FILENAME, std::ios::binary);
47      if (!ifs)
48      {
49          std::cerr << "Failed to open communication file for receiving.\n";
50          return inbox;
51      }
52
53      while (ifs)
54      {
55          MessageHeader header;
56          ifs.read(reinterpret_cast<char *>(&header), sizeof(header));
57          if (!ifs)
58              break; // EOF or error
59
60          std::string payload(header.size, '\0');
61          ifs.read(&payload[0], header.size);
62
63          if (header.receiver == my_pid)
64          {
65              inbox.emplace_back(header.sender, payload);
66          }
67      }
68
69      ifs.close();
70      return inbox;
71  }
```

ipc (using a common file in the file system to pass messages)

Driver for ipc w/ file example

1	Parent	Clear communication file (truncate to empty).
2	Parent	Forks → new child process created.
3	Parent	Parent **immediately** sends a message to the child.
4	Child	Child **waits 1 second** (`sleep(1)`), then sends a message back.
5	Parent	Waits for child to finish (`waitpid`).
6	Parent and Child	Each reads messages for themselves.
7	Parent	Deletes the communication file after done.

```cpp
int main()
{
    pid_t my_pid = getpid();

    // Clear communication file at start
    std::ofstream ofs(COMM_FILENAME, std::ios::binary | std::ios::trunc);
    ofs.close();

    pid_t child_pid = fork();

    if (child_pid == 0)
    {
        // Child process
        pid_t child_pid = getpid();
        sleep(1); // Let parent send first

        send_message(child_pid, my_pid, "Hello from child!");

        auto inbox = receive_messages(child_pid);
        for (const auto &[sender, msg] : inbox)
        {
            std::cout << "Child received from PID " << sender << ": " << msg << "\n";
            check_message(msg, "Hello from parent!");
        }

        _exit(0);
    }
    else
    {
        // Parent process
        send_message(my_pid, child_pid, "Hello from parent!");

        int status;
        waitpid(child_pid, &status, 0); // wait for child
        std::cout << "Child process finished.\n";

        auto inbox = receive_messages(my_pid);
        for (const auto &[sender, msg] : inbox)
        {
            std::cout << "Parent received from PID " << sender << ": " << msg << "\n";
            check_message(msg, "Hello from child!");
        }

        unlink(COMM_FILENAME.c_str());
    }
    // Clean up communication file
    remove(COMM_FILENAME.c_str());
    return 0;
}
```

Kenneth J. Roche
University of Washington

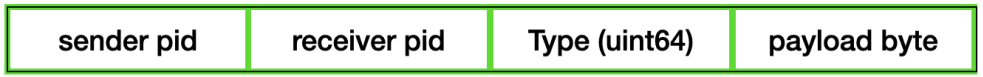
- *ipc (using a common file in the file system to pass messages)*

- First: a `MessageHeader` struct
  - 1000 (sender PID, 4 bytes)
  - 1001 (receiver PID, 4 bytes)
  - 17 (size, 8 bytes)
- Then: the payload
  - "Hello from parent!" (17 bytes)

```
bash-3.2$ g++ -std=c++17 -o xfile-messages file-messages.cpp
bash-3.2$ ./xfile-messages
Child received from PID 91611: Hello from parent!
✅ Message matches expected.
Child process finished.
Parent received from PID 91612: Hello from child!
✅ Message matches expected.
bash-3.2$
```

The file now contains **this**:

Offset	Data	Size
0	Sender PID = 1000	4 bytes
4	Receiver PID = 1001	4 bytes
8	Payload size = 17	8 bytes
16	Payload = "Hello from parent!"	17 bytes




```cpp
void check_message(const std::string &actual, const std::string &expected)
{
    if (actual == expected)
    {
        std::cout << "✅ Message matches expected.\n";
    }
    else
    {
        std::cout << "❌ Message mismatch!\nExpected: " << expected << "\nActual: " << actual << "\n";
    }
}
```

Kenneth J. Roche
University of Washington

- *ipc (using mmap in a file to pass messages as shared communication)*


```cpp
#include <iostream>
#include <vector>
#include <string>
#include <cstring>
#include <fcntl.h>      // open
#include <sys/mman.h>   // mmap, munmap
#include <sys/stat.h>   // fstat
#include <sys/types.h>
#include <unistd.h>     // close, fork, getpid, sleep, unlink
#include <sys/wait.h>   // waitpid
#include <cassert>

const std::string COMM_FILENAME = "comm_mmap.dat";

struct MessageHeader
{
    pid_t sender;
    pid_t receiver;
    size_t size;
};
```

- *ipc (using mmap in a file to pass messages as shared communication)*

```
void *mmap(void *addr, size_t length, int prot, int flags, int fd, off_t offset);
```

Parameter	Meaning
`addr`	**Hint** address. Usually `nullptr` to let the system pick any free memory address.
`length`	Size (in bytes) of mapping (how much memory you want).
`prot`	**Protection**: read/write access (`PROT_READ` , `PROT_WRITE` , etc.).
`flags`	**Sharing behavior**: `MAP_SHARED` (changes are visible to others) or `MAP_PRIVATE` (copy-on-write).
`fd`	File descriptor of the file to map.
`offset`	Where to start mapping inside the file (typically 0 for full file).

the **file** is **directly mapped into memory**!

- Writing to `addr` updates the file.
- Reading from `addr` reads from the file.
- It's super fast and handled by the OS!

- *ipc (using mmap in a file to pass messages as shared communication*

Writing (`send_message`)

- Open or create the communication file.
- Find **current size**.
- Expand the file with `ftruncate`.
- `mmap` the entire file for read/write.
- Jump to the **end** (`old_size` offset) and write:
  - `MessageHeader`
  - Then the payload (`message`)
- `munmap` the memory and close the file.

```cpp
void send_message(pid_t sender, pid_t receiver, const std::string &message)
{
    int fd = open(COMM_FILENAME.c_str(), O_RDWR | O_CREAT, 0666);
    if (fd == -1)
    {
        perror("open");
        exit(1);
    }

    size_t msg_size = sizeof(MessageHeader) + message.size();

    // Find current file size (to append at end)
    struct stat st;
    fstat(fd, &st);
    size_t old_size = st.st_size;
    size_t new_size = old_size + msg_size;

    // Expand the file
    if (ftruncate(fd, new_size) == -1)
    {
        perror("ftruncate");
        exit(1);
    }

    // Map the new space
    void *addr = mmap(nullptr, new_size, PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);
    if (addr == MAP_FAILED)
    {
        perror("mmap");
        exit(1);
    }

    // Write the message at the old end
    char *write_ptr = static_cast<char *>(addr) + old_size;

    MessageHeader header{sender, receiver, message.size()};
    std::memcpy(write_ptr, &header, sizeof(header));
    std::memcpy(write_ptr + sizeof(header), message.data(), message.size());

    // Cleanup
    munmap(addr, new_size);
    close(fd);
}
```

```cpp
void *memcpy(void *dest, const void *src, size_t count);
```

Parameter	Meaning
`dest`	Destination address (where you want to copy to).
`src`	Source address (what you want to copy from).
`count`	Number of bytes to copy.

Kenneth J. Roche
University of Washington
AMATH 483 / 583 High Performance Scientific Computing
Spring Quarter 2025

- *ipc (using mmap in a file to pass messages as shared communication)*

Reading (`receive_messages`)

- Open file for reading.
- Map entire file with `mmap`.
- Start from beginning and **walk** through the file:
  - Read a `MessageHeader`
  - Then read the payload.
- Only keep messages where `receiver == my_pid`.
- Clean up (`munmap`, close).

```cpp
std::vector<std::pair<pid_t, std::string>> receive_messages(pid_t my_pid)
{
    std::vector<std::pair<pid_t, std::string>> inbox;

    int fd = open(COMM_FILENAME.c_str(), O_RDONLY);
    if (fd == -1)
    {
        perror("open");
        return inbox;
    }

    struct stat st;
    fstat(fd, &st);
    size_t file_size = st.st_size;
    if (file_size == 0)
    {
        close(fd);
        return inbox; // nothing to read
    }

    void *addr = mmap(nullptr, file_size, PROT_READ, MAP_SHARED, fd, 0);
    if (addr == MAP_FAILED)
    {
        perror("mmap");
        close(fd);
        return inbox;
    }

    const char *read_ptr = static_cast<const char *>(addr);
    const char *end_ptr = read_ptr + file_size;
    while (read_ptr < end_ptr)
    {
        MessageHeader header;
        std::memcpy(&header, read_ptr, sizeof(header));
        read_ptr += sizeof(header);

        std::string payload(header.size, '\0');
        std::memcpy(&payload[0], read_ptr, header.size);
        read_ptr += header.size;

        if (header.receiver == my_pid)
        {
            inbox.emplace_back(header.sender, payload);
        }
    }

    munmap(addr, file_size);
    close(fd);
    return inbox;
}
```

Kenneth J. Roche
University of Washington

- *ipc (using mmap in a file to pass messages as shared communication)*

```
int stat(const char *path, struct stat *buf);
```

```
struct stat {
    dev_t     st_dev;     // ID of device containing file
    ino_t     st_ino;     // Inode number
    mode_t    st_mode;    // File type and mode (permissions)
    nlink_t   st_nlink;   // Number of hard links
    uid_t     st_uid;     // User ID of owner
    gid_t     st_gid;     // Group ID of owner
    dev_t     st_rdev;    // Device ID (if special file)
    off_t     st_size;    // Total size, in bytes
    blksize_t st_blksize; // Blocksize for filesystem I/O
    blkcnt_t  st_blocks;  // Number of 512B blocks allocated

    struct timespec st_atim; // Time of last access
    struct timespec st_mtim; // Time of last modification
    struct timespec st_ctim; // Time of last status change
};
```

```
66  std::vector<std::pair<pid_t, std::string>> receive_messages(pid_t my_pid)
67  {
68      std::vector<std::pair<pid_t, std::string>> inbox;
69
70      int fd = open(COMM_FILENAME.c_str(), O_RDONLY);
71      if (fd == -1)
72      {
73          perror("open");
74          return inbox;
75      }
76
77      struct stat st;
78      fstat(fd, &st);
79      size_t file_size = st.st_size;
80      if (file_size == 0)
81      {
82          close(fd);
83          return inbox; // nothing to read
84      }
85
86      void *addr = mmap(nullptr, file_size, PROT_READ, MAP_SHARED, fd, 0);
87      if (addr == MAP_FAILED)
88      {
89          perror("mmap");
90          close(fd);
91          return inbox;
92      }
93
94      const char *read_ptr = static_cast<const char *>(addr);
95      const char *end_ptr = read_ptr + file_size;
96      while (read_ptr < end_ptr)
97      {
98          MessageHeader header;
99          std::memcpy(&header, read_ptr, sizeof(header));
00          read_ptr += sizeof(header);
01
02          std::string payload(header.size, '\0');
03          std::memcpy(&payload[0], read_ptr, header.size);
04          read_ptr += header.size;
05
06          if (header.receiver == my_pid)
07          {
08              inbox.emplace_back(header.sender, payload);
09          }
10      }
11
12      munmap(addr, file_size);
13      close(fd);
14      return inbox;
15  }
```

- *ipc (using mmap in a file to pass messages as shared communication)*

```
bash-3.2$ g++ -std=c++17 -o xmmap-messaging kr-mmap-messaging.cpp
bash-3.2$ ./xmmap-messaging
Child received from PID 92301: Hello from parent!
Child process finished.
Parent received from PID 92306: Hello from child!
bash-3.2$ ls *.dat
ls: *.dat: No such file or directory
bash-3.2$
```

```cpp
void check_message(const std::string &received, const std::string &expected)
{
    assert(received == expected);
}

int main()
{
    pid_t my_pid = getpid();

    // Clear communication file
    int fd = open(COMM_FILENAME.c_str(), O_RDWR | O_CREAT | O_TRUNC, 0666);
    close(fd);

    pid_t child_pid = fork();

    if (child_pid == 0)
    {
        // Child process
        pid_t my_pid = getpid();
        sleep(1); // Let parent send first

        send_message(my_pid, getppid(), "Hello from child!");

        auto inbox = receive_messages(my_pid);
        for (const auto &[sender, msg] : inbox)
        {
            std::cout << "Child received from PID " << sender << ": " << msg << "\n";
            check_message(msg, "Hello from parent!");
        }

        _exit(0);
    }
    else
    {
        // Parent process
        send_message(my_pid, child_pid, "Hello from parent!");

        int status;
        waitpid(child_pid, &status, 0);

        std::cout << "Child process finished.\n";

        auto inbox = receive_messages(my_pid);
        for (const auto &[sender, msg] : inbox)
        {
            std::cout << "Parent received from PID " << sender << ": " << msg << "\n";
            check_message(msg, "Hello from child!");
        }

        unlink(COMM_FILENAME.c_str()); // Clean up
    }

    return 0;
}
```

```cpp
#include <iostream>
#include <thread>

void threadFunc(int id)
{
    std::cout << "Thread " << id << " started." << std::endl;
    // Do some work...
    std::cout << "Thread " << id << " finished." << std::endl;
}

int main()
{
    const int numThreads = 4;
    std::thread threads[numThreads];

    // Spawn threads
    for (int i = 0; i < numThreads; ++i)
    {
        threads[i] = std::thread(threadFunc, i);
    }

    // Wait for threads to finish
    for (int i = 0; i < numThreads; ++i)
    {
        threads[i].join();
    }

    std::cout << "All threads finished." << std::endl;

    return 0;
}
```

- C++ threads
  - OS allocates a new stack for each thread used to store local variables and function call frames
  - threads share the same heap and code segment as main()
  - dynamically allocated memory is visible to all threads
  - when threads finish, the stack is deallocated by OS, and frees memory it allocated on the heap

- *C++ threads introduction*
  - *processing in shared memory and lightweight processes (threads)*

Kenneth J. Roche
University of Washington
AMATH 483 / 583 High Performance Scientific Computing
Spring Quarter 2025

```cpp
#include <iostream>
#include <thread>

void threadFunc(int id)
{
    std::cout << "Thread " << id << " started." << std::endl;
    // Do some work...
    std::cout << "Thread " << id << " finished." << std::endl;
}

int main()
{
    const int numThreads = 4;
    std::thread threads[numThreads];

    // Spawn threads
    for (int i = 0; i < numThreads; ++i)
    {
        threads[i] = std::thread(threadFunc, i);
    }

    // Wait for threads to finish
    for (int i = 0; i < numThreads; ++i)
    {
        threads[i].join();
    }

    std::cout << "All threads finished." << std::endl;

    return 0;
}
```

```
bash-3.2$ g++ -o xcpp-thread-into -std=c++14 cpp-thread-intro.cpp
bash-3.2$ ./xcpp-thread-into
Thread 0 started.Thread
Thread 1Thread 0Thread 2 started.
3 started.Thread 2 finished.

Thread 1 finished.
 finished.
 started.
Thread 3 finished.
All threads finished.
bash-3.2$
```

End Lecture 13